On Re-engineering the X.509 PKI with Executable Specification for Better Implementation Guarantees

Joyanta Debnath
The University of Iowa
joyanta-debnath@uiowa.edu

Sze Yiu Chau
The Chinese University of Hong Kong
sychau@ie.cuhk.edu.hk

Omar Chowdhury*
The University of Iowa
omar-chowdhury@uiowa.edu

ABSTRACT

The X.509 Public-Key Infrastructure (PKI) standard is widely used as a scalable and flexible authentication mechanism. Flaws in X.509 implementations can make relying applications susceptible to impersonation attacks or interoperability issues. In practice, many libraries implementing X.509 have been shown to suffer from flaws that are due to noncompliance with the standard. Developing a compliant implementation is especially hindered by the design complexity, ambiguities, or under-specifications in the standard written in natural languages. In this paper, we set out to alleviate this unsatisfactory state of affairs by re-engineering and formalizing a widely used fragment of the X.509 standard specification, and then using it to develop a high-assurance implementation. Our X.509 specification re-engineering effort is guided by the principle of decoupling the syntactic requirements from the semantic requirements. For formalizing the syntactic requirements of X.509 standard, we observe that a restricted fragment of attribute grammar is sufficient. In contrast, for precisely capturing the semantic requirements imposed on the most-widely used X.509 features, we use quantifier-free first-order logic (QFFOL). Interestingly, using QFFOL results in an *executable specification* that can be efficiently enforced by an SMT solver. We use these and other insights to develop a high-assurance X.509 implementation named CERES. A comparison of CERES with 3 mainstream libraries (*i.e.*, mbedTLS, OpenSSL, and GnuTLS) based on 2 million real certificate chains and 2 million synthetic certificate chains shows that CERES rightfully rejects malformed and invalid certificates.

CCS CONCEPTS

• **Networks** → **Security protocols**; • **Security and privacy** → *Software security engineering*.

KEYWORDS

PKI; X.509 Certificate; Network Security; SSL/TLS Protocol; Differential Testing; Authentication; SMT Solver

*Corresponding author of this paper.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3484793

ACM Reference Format:
Joyanta Debnath, Sze Yiu Chau, and Omar Chowdhury. 2021. On Re-engineering the X.509 PKI with Executable Specification for Better Implementation Guarantees . In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3460120.3484793

1 INTRODUCTION

Many networked applications nowadays rely on the Public-Key Infrastructure (PKI) for achieving authentication guarantee. Among PKI proposals, X.509 is the most prominent one and is widely used for establishing a secure communication channel together with Transport Layer Security (TLS). It is also used in other application scenarios including, but not limited to, the signing and verification of emails and software packages. Consequently, the ability to correctly parse, process, and validate X.509 certificates is often a critical prerequisite for achieving the desired security guarantees.

The X.509 PKI is designed to be flexible in terms of the stipulation and enforcement of security policies. Specifically, version 3 of the X.509 standard, which is by far the most used version on the Internet today, introduced the concept of *certificate extensions*. Extensions enable issuers to specify and impose additional restrictions on the usage of the certificates issued by them. Moreover, apart from the standard extensions that were profiled for Internet usage, organizations and entities are free to introduce and incorporate custom-made extensions on X.509 version 3 certificates. As an example, the Google Certificate Transparency project uses a custom certificate extension as a means for distributing signed timestamps.

While in theory X.509's flexibility makes it adaptable to different application scenarios, in reality it greatly complicates both the specification of X.509 and implementations of certificate validation. Implementation flaws can thus result in failures to enforce the desired security policies stipulated by the certificate fields and extensions. In fact, flaws in implementations of certificate validation abound in practice. Previous testing efforts explored the usage of fuzzing [14, 17] and dynamic symbolic execution [15] to find bugs in common implementations of certificate validation. They found numerous instances of deviation from the specification. Many of these noncompliance instances severely threaten the expected guarantees stemmed from the correct enforcement of security policies. Such implementation flaws and non-compliant behavior are further exacerbated due to a lack of formalized specification or a reference implementation accompanying the X.509's natural language specification [19]. *In this paper, we aim to improve the status-quo by first re-engineering and formalizing a fragment of X.509's specification, and then using the formalized specification to develop a high-assurance implementation.*

Challenges. There are several challenges to our work. First, the X.509 standard is written in a natural language (English), which can be ambiguous and inconsistent. Moreover, the specification is not always explicit in whether a particular input is acceptable, especially, when requirements can be classified as "*producer rules*" (which are to be followed by the certificate issuers, as discussed in a prior work on misissuance [35]) instead of "*consumer rules*" (which need to be enforced by the certificate validation implementations). Second, prior to enforcing the semantic requirements of certificate validation, one needs to be able to parse the certificates, which requires dealing with the ASN.1 notation and its Distinguished Encoding Rule (DER). Due to its design, a grammar that parses objects encoded with DER is inherently context-sensitive [32], and as such, it is difficult to use an off-the-shelf parser generator to develop an X.509 certificate parser.

Re-engineering the specification. We attempt to re-engineer the X.509 specification to better facilitate the proper implementation of security policy enforcement. As such, we set out to manually inspect the current standard and partition the consumer rules into two categories: *syntactic requirements* and *semantic requirements*. Among them, the syntactic requirements regulate the correct format of an X.509 certificate as in the DER encoding of ASN.1. The semantic requirements, on the other hand, impose restrictions on the field values of individual certificate as well as inter-relationships that must hold between different certificate fields. Intertwined nature of these two types of requirements makes it challenging to decompose the specification. In our decomposition of the specification, we take the stance that anything specific to the DER encoding of the certificates are essentially syntactic restrictions whereas the rest are considered semantic requirements.

Formalization. Formalizing the syntactic restrictions of the DER encoding of an X.509 certificate requires a context-sensitive grammar [32]. We first used a parser combinator framework to provide an executable specification for the X.509 syntactic requirements. During this exercise, we observed that a restricted fragment of attribute grammar, is expressive enough to precisely capture these syntactic requirements. For this fragment, we provide a domain-specific language (DSL) in which one can write the X.509's syntactic requirements. We then developed a parser generator that can automatically generate parsers for a grammar written in our DSL. We want to emphasize that our DSL is sufficiently generic and expressive to capture certificate fields with complex formats, as well as other objects such as the encoded (and padded) hash digest used in PKCS#1 v1.5 RSA signatures [30]. We particularly developed this DSL to study and validate the exact expressive power needed to capture the syntactic requirements of an X.509 certificate.

For the semantic requirements of the most widely used certificate fields and extensions (guided by our measurements), we observe that these requirements are essentially assertions imposed on the decoded certificate field values and hence form an *executable specification*. We use quantifier-free first order logic (QFFOL) to capture these assertions. This choice of formalism has two advantages: (1) one can use an SMT solver to check whether the semantic restrictions are consistent and conflict-free (*i.e.*, there is at least one certificate which satisfies all the requirements); (2) it is possible to use the SMT solver to check whether a decoded X.509 certificate is specification compliant. Such an approach allows one to use the SMT solver to obtain rich diagnostic information when the certificate chain being checked does not comply with the specification.

Executable specification to CERES. Based on these and other insights, we develop a high-assurance X.509 implementation named CERES (CERtificate Executable Specification). Roughly, CERES glues together the parser (generated either from the parser-combinator specification, or the DSL-based parser generator) with the SMT-based semantic checker. In reality, to have a full-fledged implementation, we need to address additional challenges including building a chain of X.509 certificates, computing signatures, normalizing name fields, etc. For evaluating CERES, we carry out a differential testing with three widely used libraries (*i.e.*, mbedTLS, OpenSSL, GnuTLS) in terms of compliance and overhead. In our differential testing, we first use 2 million certificate chains constructed from a local snapshot of unique certificates provided by Censys in 2019. We also used 2 million certificate chains generated by a specialized X.509 fuzzer called Frankencert [14]. Our evaluation revealed that CERES is more restrictive in terms of compliance than the tested libraries while incurring around $11x$ runtime overhead (0.462 seconds compared to 0.042 seconds). The large overhead of CERES is expected as we use an SMT solver to enforce the semantic restrictions. The large overhead is not a concern as we expect CERES to be used for compliance checking purposes of other libraries instead of being directly used in the wild. Notable findings include GnuTLS and OpenSSL allowing extensions with unexpected certificate versions (*e.g.*, version 1 or 4), not enforcing exact length restrictions, and allowing relaxed bit-string encodings. GnuTLS also accepts any malformed critical extensions containing random octet strings.

Contributions. In summary, the current paper makes the following technical contributions:

(1) We re-engineered the specification of X.509 by decoupling the syntactic restrictions from their semantic counterparts.
(2) For formalizing the syntactic restrictions, we identified a fragment of attribute grammar that is sufficiently expressive. We developed a DSL which can capture grammar written in this fragment. We also developed a parser generator that can generate a parser for this fragment of attribute grammar. This allows us to obtain a high-assurance, specification-compliant X.509 certificate parser.
(3) We used the concept of *executable specification* and developed an executable specification for X.509 semantic requirements in QFFOL, which can be enforced by an SMT solver.
(4) We implemented a high-assurance implementation called CERES, and showed its effectiveness in identifying non-compliant behaviors of other libraries with differential testing. We responsibly disclosed our findings to the corresponding library developers. The implementation of CERES and its detailed documentation are publicly available at [21].

2 BACKGROUND AND RELATED WORK

2.1 X.509 and noncompliance

An X.509 certificate consists of three parts: TbsCertificate, SignatureAlgorithm, and SignatureValue; see Figure 1 for the high level structure of a certificate. The TbsCertificate part generally contains information on certificate version, unique serial number, validity period, certificate issuer name, certificate subject name (owner),

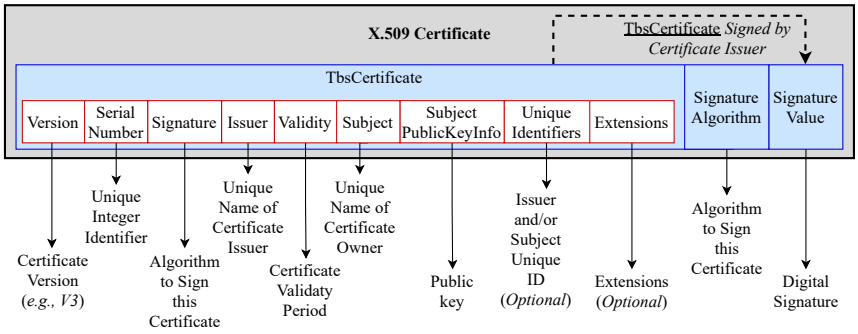

Figure 1: A typical structure of an X.509 certificate [7].

public key, the algorithm used by the issuer to sign this certificate, and few optional fields (*i.e.*, unique identifiers, a sequence of X.509 version 3 extensions). The certificate issuer signs the whole content of TbsCertificate part to generate a signature. This signature is appended at the end of the certificate as SignatureValue.

Validating a certificate chain entails the parsing and checking of certificates fields and extensions. Many previous research efforts focused on testing the certificate validation logic of TLS libraries, with the use of fuzzing [14, 17], and dynamic symbolic execution [15]. Apart from validation, issuance of X.509 certificates can also deviate from the specification [24, 35], sometimes resulting in malformed, non-compliant certificates. While numerous instances of noncompliance were found by these previous efforts, they inherently suffer from false negatives. More fundamentally, they do not prescribe an alternative approach of implementing certificate validation, which is a gap that we want to address in this paper.

2.2 Avoiding weaknesses and being compliant

Developing bug-free implementations that are compliant with the specification is a general challenge shared by many security-critical network and cryptographic protocols. Efforts have been made in implementing and formally verifying cryptographic libraries [13, 40, 49]. However, these efforts currently do not have a verified implementation of X.509 certificate validation.

Re-engineering the X.509 natural language specification has been attempted before as part of an effort to re-engineer the TLS natural language specification [31]. The OCaml source code of their TLS stack written as part of their re-engineering effort, which also includes the X.509 certificate chain validation, can be viewed as a form of executable specification. Our re-engineering and formalization efforts of X.509, however, vary from theirs in the following ways: we have (1) a clear separation of syntactic and semantic requirements; (2) identified the different formalization sufficient to precisely capture these two types of requirements; (3) formal evidence of the consistency of our formalization. We note that their code for parsing DER-encoded certificates is very similar to our parser-combinator based approach.

Recall that, capturing the ASN.1 DER encoding rules of a syntactically valid X.509 certificate requires a context-sensitive grammar [32]. There are several parser generators (with the support of context-sensitivity) [9, 22, 25, 29, 36, 38, 43] one can consider for developing a high-assurance parser for an X.509 certificate. However, all of these languages for expressing a context-sensitive grammar are more general than what is needed for X.509. As a result, even though many of them can be used for our purpose, we rely on our own DSL so that we can experiment and determine the precise expressive power needed to capture the syntactic requirements. This exercise enabled us to conclude that one does not need backtracking to parse an X.509 certificate.

There are other DSLs devised to focus on syntactic requirements similar to that of X.509 [10, 41]. Among them, Ramananandro *et al.* [41] presented Everparse which is a framework for generating provably correct, zero-copy parsers from declarative descriptions of tag-length-value (TLV) binary message formats. However, it is not clear how to encode the length constraints of X.509 certificates with Everparse's front-end. We, however, do not preclude the possibility of bypassing Everparse's front-end and instead using their parser combinator library directly to write a formally correct parser of X.509 certificates. More recently, Tao *et al.* [44] developed a formally correct and memory safe encoder of X.509 certificates in F*. Their guarantee includes being able to correctly encode an internal representation of X.509 certificates to their corresponding byte stream format in DER. For our purpose, however, we require the decoder of a certificate. Barenghi *et al.* [10] attempted to create context-free and regular specifications for X.509. However, their effort depended on major simplifications of the X.509 *grammar* and *discretizations* of few certificate fields to avoid context-sensitivity.

3 OVERVIEW OF OUR APPROACH

In this section, we present our overarching objective, scope of the work, and a high-level description of our approach.

Overarching objective. The overarching objective of this work is to facilitate developing a formally verified, RFC 5280-compliant reference implementation for X.509 certificate chain validation logic. For achieving this overarching objective, a typical workflow has the following steps: ❶ develop a formal specification for the X.509 certificate chain validation logic Φ in some suitable formalism that precisely captures the requirements prescribed in RFC 5280; ❷ check the formal specification Φ is logically consistent, that is, there does not exist two requirements that are at odds with each other; ❸ develop an implementation I that realizes all the security policy checks mandated by RFC 5280; ❹ finally, using some formal verification technique, show that I satisfies Φ.

Our current work. In this work, we focus on the steps ❶, ❷, and ❸ of the above workflow. Interestingly, after developing the formal

specification of the X.509 standard Φ, we observed that a high-assurance, albeit *not* formally verified, implementation for X.509 certificate chain validation I_{HA} follows directly from Φ, without having to manually encode the requirements of Φ in programming languages like C. This is because the resulting formal specification Φ essentially contains constraints on the different certificate fields and hence induces an *executable/operational specification*. Intuitively, given an executable specification Φ, one can use a high-assurance interpreter (*e.g.*, a constraint solver) that interprets Φ given the concrete values of the certificate fields of the input certificates (organized in a chain), and thus enforcing the constraints imposed by Φ. The resulting implementation CERES, stemmed from our specification re-engineering and formalization effort, faithfully follows and enforces the requirements given in the specification, under the assumption that the underlying constraint solver is correct, thus leading to a higher level of assurance. We envision CERES to be used in the following contexts: (1) an oracle for testing RFC compliance of a given X.509 library through differential testing [14, 15, 17]; (2) checking whether a certificate chain to be used during the configuration of a server with TLS support is RFC compliant.

Table 1: Example of syntactic and semantic requirements

Field	Syntactic Requirement	Semantic Requirement
Version	Version ::= INTEGER { v1(0), v2(1), v3(2) }	When extensions are used, as expected in this profile, Version MUST be 3 (value is 2).
KeyUsage Extension	KeyUsage ::= BIT STRING { digitalSignature (0), nonRepudiation (1), keyEncipherment (2), dataEncipherment (3), keyAgreement (4), keyCertSign (5), cRLSign (6), encipherOnly (7), decipherOnly (8) }	When the KeyUsage extension appears in a certificate, at least one of the bits MUST be set to 1.

Re-engineering the specification. Before formalizing the specification, we first re-engineer the requirements from RFC 5280 into two classes of rules: *syntactic rules* and *semantic rules*. The syntactic rules capture the obligations involved in decoding an X.509 certificate encoded under DER as a byte stream. In contrast, semantic rules put restrictions on the individual field values of a certificate, and as well as relationships that should be maintained by field values in different certificates in a given chain. A few examples of such requirements can be found in Table 1. In row 1 of this example, syntactic requirement of the `Version` field of a certificate is that it should be an integer that can take a value from the set $\{0, 1, 2\}$. An example semantic requirement involving the `Version` field states that `Version` field value should be 2 (*i.e.*, version 3 certificates are represented by the certificate version field having the value 2) when a certificate contains extensions. Note that, one can further partition the rules into two more classes: *producer rules* and *consumer rules*. Producer rules are imposed on certificate issuers whereas consumer rules are requirements on certificate validation implementations. In our formalization, we focus on consumer rules.

Separating the specification into syntactic and semantic requirements has several advantages. First, it makes the specification modular and allows extending one without impacting the other. Second, one can use two different formalism to capture the two types of requirements without having to resort to a single unifying formalism expressive enough to capture both. Finally, one can use two different types of proof techniques to discharge a modular implementation complying with the two different types of requirements. Unfortunately, the two types of requirements are often intertwined and there might not be a well-defined separation between them. At one extreme, the whole X.509 specification can be viewed as just a set of *syntactic* requirements to be enforced during the parsing/decoding. Such an extreme perspective, however, makes the parser overly complicated. At the other extreme of delegating most of the syntactic requirements to *semantic* checks, parsing can become ambiguous, as we will discuss in Section 4.

To elaborate, suppose a certificate has the form $a^n b^n$ where a, b are terminals and $n \in \mathbb{Z}_+$. This is a classic example of a context-sensitive grammar that accepts strings in which as are followed by an equal number of bs. There are two ways to decompose this certificate format into syntactic and semantic requirements, embracing the two extremes discussed above. First, one can consider the syntactic requirement to be the regular grammar a^+b^+ whereas the semantic requirement checks whether the number of as equals to the number of bs. Second, one can consider the syntactic restriction to be $a^n b^n$ with no semantic restrictions. Under the former decomposition, to parse certificates, a parser capable of parsing regular languages would suffice. However, under the latter decomposition, a parser that can parse context-sensitive languages would be needed. The sweet spot is somewhere in the middle of the two extremes. Ideally, requirements related to DER encoding are considered as syntactic requirements, and the rest are taken as semantic requirements. During our re-engineering effort, we try to stay close to the sweet spot, sometimes using the simplicity of parsing as a metric for partitioning requirements when it is not apparent.

Formalizing syntactic requirements. There are three challenges to formalize the syntactic requirements of a DER-encoded X.509 certificate. First, the current standard is written in natural languages and suffers from ambiguity and under-specification. Second, different documents (*e.g.*, RFC 4518 [48], X.690 [28]) need to be consulted to get a complete picture of the syntactic requirements. Finally, the DER encoding of an X.509 certificate is overall complex; containing sub-components that have complex structures of their own (*e.g.*, RSA signature). We first write an executable specification of the syntactic requirements using a parser combinator framework. During this exercise, we observe that a restricted fragment of attribute grammar is sufficient to capture the complex syntactic structure of a certificate. At a high-level, one can view this fragment of attribute to be enhancing the unambiguous LL(1) context-free grammar with some context for handling length restrictions of the DER encoding. Unlike full attribute grammar, parsing this restricted fragment does not require backtracking when encountering a rule with different choices. This is ensured by the unambiguous LL(1) portion of the grammar (*i.e.*, no left recursion, already left-factored). One can thus write a recursive descent parser with the support for carrying some context for checking the constraint imposed by the ℓ field. We developed a domain-specific language (DSL) for expressing grammars in this restricted fragment of attribute grammar. We developed this DSL, and the corresponding parser generator, in addition to the parser combinator based specification, to demonstrate and validate that this fragment is sufficient to express X.509 syntactic requirements. Such a precise characterization of the syntactic requirements is currently missing in the literature.

Formalizing semantic requirements. Along with the challenge of natural language specifications suffering from ambiguity and

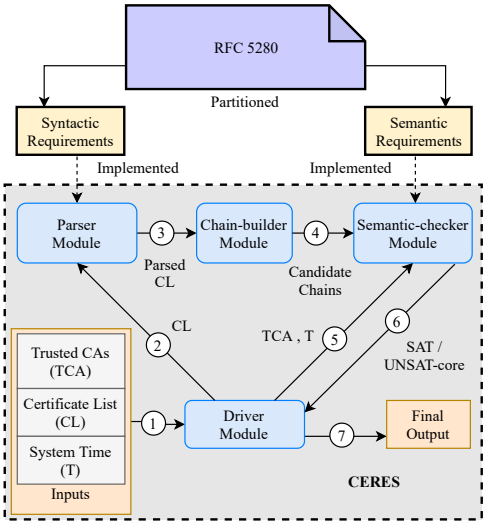

Figure 2: Realization of CERES

under-specification, an additional challenge of formalizing semantic requirements is to choose a formalization that is sufficiently expressive enough to capture the requirements and also amenable to automatic checking of the specification consistency. We observed that a quantifier-free first-order logic (QFFOL) suffices to capture the requirements. It not only allows one to automatically check the consistency of the specification but also allows one to realize a working, high-assurance implementation with support for rich diagnostics through the use of an SMT solver.

From executable specification to CERES. We now discuss how we used the above insights to realize a high-assurance implementation (step ❸ of the workflow) of the X.509 standard (see Figure 2). CERES takes a certificate chain, the current system time, and a trust anchor store (aka, CA root store) as input, and returns the certificate validation result as well as other diagnostic information as output. It is modularly realized from the following four logical pieces: Parser, Chain-builder, Driver, and Semantic-checker. The Parser takes as input the certificate chain to be validated as well as the trust anchor store, and returns the parse trees corresponding to the certificates. The Chain-builder module takes these parsed input certificates and forms candidate certificate chains. The Semantic-checker then takes as input the current time, the semantic rules corresponding to the standard in QFFOL, the ASTs corresponding to a candidate certificate chain and the certificates in the trust anchor store, and then communicates with the SMT solver to check the assertions enforced by the semantic requirements as well as collect diagnostic information. The Driver does the plumbing needed to combine the Parser and the Semantic-checker. It has multiple functions: (a) it calls the Parser component with the right input; (b) it then calls the Chain-builder with the input parsed certificates to form candidate chains; (c) it converts the parse trees organized in a candidate chain returned by the Chain-builder to the corresponding ASTs; (d) it then normalizes inputs (*e.g.*, canonicalizing Unicode strings) and perform auxiliary functionality to make the ASTs ready to be consumed by the Semantic-checker; (e) it then calls the Semantic-checker with the right input; (f) finally, it parses the outputs of the Semantic-checker and returns them as diagnostic information.

4 ENFORCING SYNTACTIC REQUIREMENTS

Here we discuss the syntactic rules that an X.509 certificate is supposed to follow, as well as how we formalize and enforce them.

4.1 Syntactic requirements from ASN.1 DER

While RFC 5280 [19] outlines some syntactic and semantic requirements that X.509 certificates have to abide by if they are to be used in the Internet as discussed in previous work [14, 15], we note that RFC 5280 alone is not the only source of compliance requirements. In most usage scenarios, including those covered by RFC 5280, an X.509 certificate is encoded using the X.690 ASN.1 Distinguished Encoding Rules (DER) [28], and hence the byte stream representing a certificate has to be compliant to the syntactic rules mandated by X.690. X.690 specification first outlines the so-called Basic Encoding Rules (BER), and DER can be seen as a more restrictive form of BER, which imposes additional limitations on encoding options. The rules thus outline how one should encode and decode the different data types used in ASN.1, including *constructed* types like SEQUENCE and SEQUENCE OF, SET and SET OF (the collection is allowed to be empty with the OF suffix), as well as *primitive* types like OBJECT IDENTIFIER, INTEGER (two's complement), BOOLEAN, BIT STRING and OCTET STRING, all of which are frequently used by X.509 to define its certificate fields and extensions. As a side note, DER requires both to be encoded in the *primitive* form, while BER also allows the *constructed* form at the option of the sender. We instead follow the ASN.1 DER given in X.690 for parsing and decoding certificate fields and extensions. For simplicity, we refer the reader to the corresponding documents (*e.g.*, [28]) for the low-level details.

4.2 Syntactic requirements from RFC 5280

On top of the requirements stemmed from ASN.1 DER, for certificates that are for use in the Internet, RFC 5280 imposes additional restrictions. As an example, for the certificate validity field, which is used to denote the time period for which the certificate can be considered as valid, one can choose to use the ASN.1 Type GeneralizedTime (another option being UTCTime). According to DER, it is possible to include fractional seconds in a representation of the type GeneralizedTime, however, RFC 5280 requires a more restrictive form of GeneralizedTime where fractional seconds are disallowed. We thus in general take the ASN.1 DER as the baseline requirements, and then impose further syntactic restrictions based on RFC 5280 if they exist.

4.3 Complexity of the syntactic requirements

One key challenge of identifying the right formalization worth highlighting is that the DER encoded byte stream forms a tree of *Tag-Length-Value* (TLV) nodes, where explicit lengths are given to describe the size of the *value* bytes that needs to be parsed, and the value bytes can be recursively defined by more TLV nodes. While on a first glance this might seem to help the consumer in processing the input, there are several reasons to why this actually increases the complexity of the parser. First, blindly trusting the provided length

value is not an acceptable practice, as the sender could potentially be malicious, and absurd length values could be abused to trick the consumer into performing a buffer over-read (akin to the infamous OpenSSL heartbleed bug, as well as the Denial-of-Service attack through a malicious RSA signature described in a recent work [16]). In other words, the length values themselves need to go through some sanity checks to make sure that they are not going beyond the size of the input certificate. Second, when dealing with the tree of TLV, the length of a parent node is obtained by summing up the size of the tag, length, and value fields of all its immediate child nodes. As such, in order to make sure the input certificate is properly encoded, during parsing one needs to keep track of the lengths of the children node, and then compute and check that the length values of the children nodes add up to the length given as input in the parent node. This is one of the contributing factors to the grammar of X.509 certificates being context-sensitive [32].

One may be tempted to delegate some of the context-sensitive aspects of the requirements, especially, the one with the length field, to the semantic requirement. In this approach, the grammar Γ will be more permissive than the standard permits whereas the accompanying semantic requirement will rule out the the spurious certificates accepted by Γ but not by the standard. A certificate has the form $\langle t, \ell, v\rangle$ where ℓ signifies the length (in bytes) of the value in v. Even under the simplifying assumption that t (not a constructed type) and ℓ are both single byte values, we can represent a certificate as the following permissive grammar: $\Gamma ::= \langle\text{byte}\rangle\langle\text{byte}\rangle\langle\text{byte}\rangle^*$. Such a permissive grammar is ambiguous and may not be able to parse complex cases where the v field contains other $\langle t, \ell, v\rangle$ triples. For a sequence of TLVs, the first value field in the TLV sequence due to the presence of the wild card character will end up consuming all the bytes only to identify the problem at the semantic check level. To address this, one would need to backtrack to try other possible derivations, which will be really inefficient.

4.4 Formalizing syntactic requirement

We start our formalization by expressing the syntactic requirements in an input language of a parser combinator framework. This framework allows one to combine multiple small parsers with some well-defined combinators (*e.g.*, choice, count, many) to modularly generate complex parsers. As an example, a parser for a grammar $\Gamma ::= A \mid B$ can be written by combining the parsers for non-terminals A and B with a choice combinator. One can keep decomposing the higher level parser as combinations of lower-level parsers. A parser written in this framework thus closely follows the syntactic requirements of a grammar and consequently can also serve as a formal specification. During this exercise, we realized that X.509 syntactic requirements do not necessarily need to use the rich combinators available in this framework. This raised to two questions: (1) How much expressive power does one need to express X.509 syntactic requirements? (2) Is it possible to avoid backtracking during parsing X.509 certificates?

Our investigation revealed that a restricted fragment of *attribute grammar* [8, 33] is sufficient to formalize our syntactic requirements. The fragment can be viewed as an extension of unambiguous LL(1) context-free grammar with some limited context. Roughly, this is similar to enhancing the Backus-Normal Form (BNF) notation for context-free grammar to allow production rules of the form: $A, \vec{c} ::= \alpha_1\beta_1\gamma_1 : \vec{r_1}\ [\psi_1, \ldots, \psi_n] \mid \alpha_2\beta_2\gamma_2 : \vec{r_2}\ [\pi_1, \ldots, \pi_n]$ in which A is a non-terminal, $\vec{c}$ is a sequence of parameters representing the input context to be used by the sequence of terminals/non-terminals denoted with $\alpha_i\beta_i\gamma_i$, $\vec{r_i}$ represents the sequence of computed context to be passed to any non-terminal production rules that are used to define A, and, $\psi_1, \ldots, \psi_n$ and $\pi_1, \ldots, \pi_n$ are constraints that must be maintained for a string to be accepted. Note that, we require A to be an unambiguous LL(1) grammar (*i.e.*, no left recursion, already left-factored) without the context. We have defined a domain-specific language (DSL) in which one can write the above restricted context-sensitive grammars (see Appendix A). Our DSL supports both predefined *synthesized* and *inherited* attributes for handling the length constraints in a TLV construct.

4.5 Syntactic requirements to parsing

Realizing a parser from our parser combinator description is straightforward. However, for our parser obtained from the custom parser generator for our DSL uses a recursive descent parser with limited context for handling the length constraints. Our recursive descent parser is very similar to a parser for any unambiguous LL(1) grammar, additionally enhanced with contexts regarding the length constraints. Our parser does not require backtracking due to the requirement that without the context and restrictions the resulting LL(1) grammar is unambiguous. Given a production $A, \vec{c} ::= B : \vec{r_1}\ [\psi_1^1, \ldots, \psi_p^1] \mid C : \vec{r_2}\ [\psi_1^2, \ldots, \psi_q^2] \mid D : \vec{r_3}\ [\psi_1^3, \ldots, \psi_r^3]$, one needs to backtrack if one cannot unambiguously decide which choice (*i.e.*, B, C, D) to explore for a successful parsing. Our LL(1) base ensures that this is not the case. Just by looking ahead one token one can unambiguously decide which branch to consider.

5 ENFORCING SEMANTIC REQUIREMENTS

For semantic requirements, we mainly consult RFC 5280, which profiles X.509 version 3 certificates for use in the Internet, and is taken as the basis of identifying non-compliant behaviors in previous work [14, 15]. Since RFC 5280 is written in natural language, it is prone to inconsistency, ambiguity, and misinterpretation. However, we make our best effort in understanding and interpreting its requirements. We give a few examples of inconsistency, ambiguity, and under-specification from RFC 5280 in Appendix B.

5.1 Certificate extensions to support

RFC 5280 [19] defines 15 standard extensions and 2 private extensions. An X.509 certificate may contain other private extensions not included in RFC 5280. Thus, the number of possible extensions to parse and interpret is unbounded. To make this manageable, we perform an analysis on 1.2 billion certificates from a static snapshot of the Censys dataset [23], which reveals that only 10 extensions appear frequently across different certificates (blue-colored extensions in Table 2). Thus, we focus on the requirements of these 10 extensions, and for other extensions, we only consume corresponding bytes to continue parsing other fields but do not enforce their associated semantic rules.

Table 2: Frequencies of extensions in Censys dataset

Extension	Freq.	Perc.	Extension	Freq.	Perc.
Basic Constraints	1,182,963,794	95.85%	Issuer Alternative Names	1,577,915	0.12%
Authority Key Identifier	1,179,639,634	95.57%	Subject Directory Attributes	14,881	0%
Subject Alternative Name	1,172,888,944	95.03%	Name Constraints	7,600	0%
Subject Key Identifier	1,170,590,756	94.85%	Freshest CRL	6,587	0%
Key Usage	1,155,599,607	93.63%	Policy Constraints	451	0%
Extended Key Usage	1,151,884,357	93.33%	Policy Mapping	347	0%
Authority Information Access	1,141,133,734	92.46%	Subject Information Access	337	0%
Certificate Policy	1,138,776,440	92.27%	Inhibit Policy	253	0%
CRL Distribution Points	278,689,659	22.58%			

5.2 Insight of executable X.509 specification

Generally speaking, a formal specification can come in two flavors: *non-operational* and *operational/executable*. An example of a non-operational specification could be of a sorting algorithm, which states that given a non-empty array of integers, the output of the sorting algorithm is a permutation of the original input array which is sorted. Such a non-operational specification expresses the requirements of a sorting algorithm without saying how to sort an array. On the contrary, an executable specification not only expresses the requirements but also (explicitly) say how to attain the desired requirement. Let us take the example of the $\max(a, b)$ function where $a, b \in \mathbb{Z}$. An operational specification of a correct max function can be $\forall a, b.\ \max(a, b) = \text{ITE}(a > b, a, b)$ where ITE is the if-then-else construct. While a correct implementation directly follows from the operational specification of the $\max(a, b)$ function, this is not the case for the sort algorithm specification.

When supporting the most widely used features of X.509 standard (shown in Section 5.1), we observed that the imposed semantic requirements are essentially assertions on individual certificate field values as well as relationships that multiple fields from different certificates should maintain. Such assertions essentially form an operational/executable specification. We use QFFOL to represent X.509's semantic requirements, which has the following two benefits. (B1) One can check the consistency of the specification by posing a query to the SMT solver to check whether the formal specification is satisfiable (fulfilling the obligation of step ❷). Being able to check formal specification's consistency allows one to identify conflicts due to ambiguous or under-specification in the standard akin to ones discussed in Appendix B. (B2) One can reduce the checking of compliance with the semantic rules to a satisfiability query to the SMT solver whose unsatisfiability entails noncompliance with the specification. In addition, when a certificate chain violates the semantic requirements, one can identify the cause of violation (*i.e.*, the violated RFC clause) and also get a proof of noncompliance for free using a proof-producing SMT solver [12].

5.3 Formalizing X.509 semantic rules

We start this section by providing the interface of an implementation enforcing X.509 certificate chain validation. We recognize that existing libraries do not always share a common interface but we present this interface for the sake of the following discussion. We assume an implementation I_{HA} takes the following inputs: (1) A certificate chain to be validated; (2) A list of X.509 certificates in its trust-anchor store; (3) The current system time. It then returns a quadruple of the form $\langle \mathsf{d}, \mathsf{pk}, \mathsf{cause}, \mathsf{proof} \rangle$ in which d denotes the certificate chain validation result (*i.e.*, `valid` or `invalid`), pk is the public key corresponding to the leaf certificate, and when d is invalid, cause denotes the cause of noncompliance (*e.g.*, violated RFC clause) and proof contains the proof of noncompliance when d is invalid. The formal specification of the partial-correctness (without termination) of an implementation I_{HA} thus should have the following form: $\forall \vec{cc}, \vec{rt}, t.\quad I_{\text{HA}}(\vec{cc}, \vec{rt}, t) = \langle \text{valid}, \_, \_, \_ \rangle \Leftrightarrow \Phi[\vec{cc}, \vec{rt}, t]$ in which $\vec{cc}$ is the certificate chain to be validated, $\vec{rt}$ is the list of the trust anchor certificates, t is the current system time, and '_' denotes an ignore value. The formula $\Phi[\vec{cc}, \vec{rt}, t]$ entails the semantic rules captured as a QFFOL formula in our context. We use the notation $\Phi[x, y]$ to denote a formula Φ in which variables x and y are free.

QFFOL theories used in our specification. When capturing the semantic rules of X.509 specification, instead of declaring a structure (*e.g.*, algebraic data type or record type) for a certificate comprising of different fields, we end up flattening the certificate and modeling each field of a certificate separately. This choice is essentially to make the form of the constraints simpler and human-readable. In addition, we do not model all the fields of the certificate, and instead capture the fields which appear in at least one semantic rule. Among the many different logic (*i.e.*, a consequence of the theories supported by an SMT solver) in an SMT solver, we use the following theories: *array*, *uninterpreted function*, *fix-width bit-vector*, and *linear integer array*. The exhaustive list of semantic rules that we capture can be found in the Appendix C. In our formalization, we use fixed-width bit-vector to represent fixed-size bit-strings whose sizes are known at static time, integer arrays with length constraints to represent sets and sequences in the ASN.1 domain, and for the rest we use logical integers supported by SMT solvers.

One of the unique aspects of our formalization is that we heavily rely on the integer data-type in an SMT solver. Unlike machine integers (32-bit and 64-bit), integers in SMT are logical integers, which have infinite precision. This allows us to represent many variable-length data-types that are perceivable as a sequence of bytes with logical integers. As an example, a canonicalized (certificate issuer/subject) name can be viewed as an array of bits/bytes and hence can be encoded as a pair of integers $\langle val, len \rangle$ in which *val* represents the integer value corresponding to the binary string whereas *len* represents the length of the array. Although it may be tempting to encode names as strings (a theory supported by major SMT solvers), the semantic requirement essentially warrant checking whether two names are equal, which does not require any theory-of-strings-specific operations. In addition, modern SMT solvers cannot generate proofs of unsatisfiability when the theory of strings and algebraic data types are used in an SMT query. Thus, to check the equality of two names s_1 (encoded as $\langle v_1, \ell_1 \rangle$) and s_2 (encoded as $\langle v_2, \ell_2 \rangle$) in our formalization we essentially check whether $v_1 = v_2 \wedge \ell_1 = \ell_2$. Checking just the values here is not sufficient because (bit-)strings of two different lengths may have the same integer value. Finally, during our formalization, we delegate constraints that require computation such as modular exponentiation necessary for signature verification, and canonicalizing Unicode strings appearing in subject/issuer name fields to dedicated helper modules that perform the actual computations.

Example. As an example (see Table 1), suppose a certificate has a mandatory field (*i.e.*, `Version` of type integer) and an optional field called `KeyUsageExtension`. Also, the `KeyUsageExtension` field has the type $\text{Bool} \times \text{BV}_9$ in which the first element of the

pair (denoted with Boolean variable KUE_isPresent) represents whether the extension is present, and the second element of the pair (denoted with BV_9 type variable KUE_data) is a bit-vector of size 9. One can formalize the semantic rules in Table 1 as formula Π[Version, KUE_isPresent, KUE_data] in the following way: (Version = 0 $\vee$ Version = 1 $\vee$ Version = 2) $\wedge$ (KUE_isPresent $\rightarrow$ (Version = 2 $\wedge$ KUE_data $\neq$ #*b*000000000)). Note that, a certificate `Version` field containing the value 2 suggests a version 3 certificate.

Consistency of our specification. After encoding the semantic rules as a QFFOL formula, we can use an SMT solver to check whether the formula is satisfiable. If the formula is satisfiable, then it suggests that there are no contradictory requirements in the specification; that is, there is at least one certificate chain that satisfies all the constraints. However, there is a challenge to achieving it. Our specification $\Phi[\vec{cc}, \vec{rt}, t]$ is parametric to the certificate chain length (*i.e.*, $|\vec{cc}|$) and also the number of certificates in the trust anchor store (*i.e.*, $|\vec{rt}|$). For checking consistency, we thus have to instantiate these two parameters before we can query an SMT solver. We vary the certificate chain length from 1 to 10, and consider one symbolic certificate in the trust anchor store. We observed that the specification is consistent for chain lengths 1 . . . 10. Although theoretically this leaves the possibility that a conflict may exist when chains of longer lengths are encountered, in practice, however, certificate chain lengths that are longer than 5 are rare.

5.4 Executable specification to implementation

Due to the nature of our executable specification, it is possible to use an SMT solver as a high-assurance interpreter to enforce the semantic requirements. Without loss of generality, suppose a certificate only has two fields and two semantic requirements as shown in Table 1. Recall that, we formalized these two semantic requirements as a QFFOL formula Π[Version, KUE_isPresent, KUE_data]. Now, given a certificate c to check for compliance after we finish parsing c, we obtained the following: c.Version $\mapsto$ 2, c.KUE_isPresent $\mapsto$ true, and c.KUE_data $\mapsto$ #*b*000000001. To check whether c is compliant with the specification Π[Version, KUE_isPresent, KUE_data], we just have to consult the SMT solver to check whether the following formula is satisfiable: Π[Version, KUE_isPresent, KUE_data] $\wedge$ Version = 2 $\wedge$ KUE_isPresent = true $\wedge$ KUE_data = #*b*000000001. If the formula is satisfiable, then we can conclude that c satisfies the semantic requirements; otherwise, it is non-compliant.

Generalizing from the example, to check whether a parsed certificate chain $\vec{C}$ is compliant with our formalization of the semantic rules $\Phi[\vec{cc}, \vec{rt}, t]$ at time t_{cur} with respect to the root store $\vec{r_s}$, we assert that the following formula is satisfiable: $\Phi[\vec{cc}, \vec{rt}, t] \wedge \vec{cc} = \vec{C} \wedge t = t_{\text{cur}} \wedge \vec{rt} = \vec{r_s}$. If the formula is satisfiable, we conclude that $\vec{C}$ is compliant; otherwise, it is non-compliant. Note that, our original specification is parametric to the lengths of $\vec{cc}$ and $\vec{rt}$. After parsing as we know the length of the certificate chain and the size of the trusted root certificate store, we can essentially instantiate our specification for those values, allowing us to avoid issuing the costly query of checking the satisfiability of a quantified formula.

5.5 Diagnostic information

Using an SMT solver to interpret the specification has the advantage that we can essentially obtain the cause of noncompliance in the form of a violated RFC clause for free. For this, we use the unsatisfiable core [18] returned by the SMT solver when the queried QFFOL formula turns out to be unsatisfiable. The unsatisfiable core is a subformula of the original queried formula which is unsatisfiable. Before issuing the satisfiability query to the SMT solver, we label each assertion (*i.e.*, semantic rule) of the formula with a semantically meaningful name. The SMT solver returns the unsatisfiable core based on these labels which in turn can be mapped back to a meaningful error message indicating the violated RFC clauses.

Note that, for faithfully checking the compliance of a certificate chain, we make the assumption that the SMT solver is correct. This is, however, not necessarily a valid assumption [46, 47]. For higher assurance, we rely on a proof-producing SMT solver [12] which can generate a proof of unsatisfiability. However, these SMT solvers cannot generate proofs for arbitrary theories. We thus choose only those theories in our formalization for which the SMT solver can generate proofs, as discussed in Section 5.3. One can also use this generated proof as a given certificate's proof of noncompliance.

6 IMPLEMENTATION

Our implementation of CERES consists of four modules: Driver, Parser, Chain-builder, and Semantic-checker, which contains around 9K lines of Python (v3) code in total. In this section, we discuss implementation details of these modules as well as their relationships.

6.1 Driver module

The Driver module requires a PEM or CRT format X.509 certificate chain as input and also allows users to pass additional inputs through command line arguments. The list of supported command line arguments are shown in Table 3. At a high level, this module initially performs some pre-processing on the input certificate chain and trusted CA certificates; particularly, each certificate is converted to DER format. Then, it calls the Parser module to parse each DER certificate of the input chain, and upon successful parsing, it attempts to build *candidate* certificate chains using the Chain-builder module. Finally, each candidate certificate chain is sent to the Semantic-checker module for semantic consistency (compliance) check. If at least one of these candidate chains pass this check, the Driver module returns validation *success* message to user; otherwise, it returns the reason for validation *failure*.

Table 3: List of supported inputs

Short Description	Argument	Default Value
Input chain location	input	None
Trusted CA store location	ca-store	Linux's CA store
Check certificate purpose	check-purpose	Any Purpose
Check UNSAT proof	check-proof	False
Check specification SAT	check-spec	False
Enable PG-based parser	dsl-parser	False
Show certificates	show-chain	False
Show current CERES version	show-version	False

6.2 Parser module

As we have discussed earlier, we follow two different approaches to implement the X.509 certificate Parser: *Parser-Generator based* (PG-based) approach and *Parser-Combinator based* (PC-based) approach, where both provide functionally equivalent parsers. In PG-based approach, we utilize a widely used parser-generator ANTLR (v4) [39]

to generate the parser for our *DSL*. We then parse the X.509 certificate grammar encoded in our DSL leveraging this DSL parser, which outputs an X.509 certificate parse tree. Finally, we use our parse tree visitor code to visit the certificate parse tree and carefully generate certificate parser code from each node of that parse tree. *In contrast*, we use the Parsec (v3.8) [6] parser-combinator for PC-based approach. The Parser module written in this approach is more concise and faster; therefore, the Driver module, by default, selects the PC-based parser and leaves the selection of PG-based parser to user's choice via a command line argument (*e.g.*, dsl-parser).

6.3 Chain-builder module

An X.509 certificate chain contains a leaf certificate and may optionally contain one or more auxiliary certificates (*e.g.*, from intermediate and root CAs). Unfortunately, certificates in an input chain can be organized out-of-order. A chain may also lack the necessary CA certificates, or contain unnecessary certificates. We thus use the Chain-builder module to build *candidate* chains that can potentially be validated after parsing. Our Chain-builder module handles both out-of-order and superfluous certificates based on the KeyIdentifier (keyid) field of the Authority Key Identifier (AKI) and Subject Key Identifier (SKI) extensions. RFC 5280 Sections 4.2.1.1–4.2.1.2 mandate the presence of AKI.keyid on conforming certificates and SKI.keyid on conforming CA certificates. Unsurprisingly, our measurement on 1.2 billions certificates from Censys dataset found 95.58% and 94.85% certificates contain the AKI and SKI extensions respectively, and in 99.99% cases AKI contains the keyid field.

We follow two rules to order *out-of-order* auxiliary certificates: (1) AKI.keyid of a certificate is identical to SKI.keyid of its issuer CA certificate, and (2) AKI.keyid is identical to SKI.keyid in a self-signed CA certificate. To handle the *absence of a root CA* certificate, after ordering existing certificates with rule (1), we find out the root CA certificate from the trusted CA store with a SKI.keyid, matching the AKI.keyid of the last intermediate CA certificate. Following this approach, Chain-builder module may end up building more than one candidate chain if multiple certificates contain the same SKI.keyid. In cases where ordering failed due to missing keyid or intermediate CA certificates, the Chain-builder module simply returns the parsed input certificate chain following the input order.

6.4 Semantic-checker module

The Semantic-checker module performs three tasks: (1) formalizes semantic constraints for a symbolic certificate chain, (2) builds formal model of the parsed certificate chain, and (3) calls CVC4 SMT solver [12] to check compliance of the semantic constraints. The complete list of these semantic constraints are in Tables 12 and 13 in Appendix C. We use PySMT [26] library to perform the formalization, and later translate this formalization to SMT-LIB [11] language. We finally send this SMT-LIB formalization as an input to the CVC4 SMT solver for checking compliance. In case of noncompliance, the SMT solver produces unsatisfiability core in a readable format. Additionally, when user passes the check-proof command line argument, the SMT solver generates a proof of unsatisfiability and checks the proof with the LFSC [37] proof-checker. *Note that*, PySMT itself does not provide any interface to generate unsatisfiability core or proof of unsatisfiability for the CVC4 SMT solver. Therefore, we rely on PySMT library *only* for the translation of the Python-based formalization to SMT-LIB syntax. Before invoking the external CVC4 solver, we append the required meta-code to enable generation of unsatisfiability core and proof of unsatisfiability.

6.4.1 Challenge – String comparison. To check the semantic constraint related to *name chaining*, we match the issuer distinguished name in one certificate with the subject name in its issuer CA certificate. The algorithm to match two names are defined in Section 7.1 of RFC 5280. This algorithm requires all the Strings to be preprocessed with LDAP StringPrep profile described in RFC 4518 [48].
Extending Haskell stringprep library. RFC 5280 instructs to use the six-step (*e.g.*, Transcode, Map, Normalize, Prohibit, Check Bidi, Insignificant Character Handling) String preparation algorithm from RFC 4518. Instead of implementing this algorithm from scratch, we start with a Haskell library stringprep [1], which already implemented the older version of StringPrep algorithm described in RFC 3454 [27]. However, this older algorithm does not perform two steps: *Transcode* and *Insignificant Character Handling*. Hence, we extend the stringprep library with these two additional steps and other improvements needed for the newer algorithm. This modified library is called by the Semantic-checker to pre-process Strings.

6.4.2 Challenge – Signature verification. We require to verify certificate Signature to obtain trust on its Public key. This verification process requires some complex cryptographic calculations.
Verifying only RSA signature. Our measurement on 1.2 billions certificates from the Censys dataset shows 95.15% certificates have RSA signatures, and only 4.83% have ECDSA signatures. Thus, we only support RSA signature verification in CERES at the moment. In case of other signature schemes, we abstract away signature verification with a stub function which always returns *True* to indicate verification is successful. For the RSA signature verification, first of all, we compute $X = S^e \bmod n$, where S represents the Signature of a certificate, and e, n represent public exponent and modulus of its issuer's public key. Next, we parse signature digest H_X out of X and then match whether $H_X = Hash(TbsCertificate)$. We perform cryptographic calculations such as *modular exponentiation* and *hash* using Python's hashlib library. However, we implemented the parser that parses X to extract the H_X. If X is syntactically malformed, it may lead to signature forgery attacks [34]. Therefore, our parser strictly checks the format of X [30] before extracting signature digest H_X.

7 EVALUATION AND RESULTS

In this section, we demonstrate the effectiveness of CERES in the context of X.509 certificate chain validation by comparing it against 3 widely used open-source TLS libraries (*e.g.*, mbedTLS [4], GnuTLS [3], and OpenSSL [5]).

7.1 Certificate chain dataset

Our test certificate dataset consists of 4 millions X.509 certificate chains. To build this dataset, we randomly sampled 2 millions certificates from the Censys [23] dataset. We then used an existing tool named cert-chain-resolver [2] to download the corresponding CA certificates of the sampled certificates. We also used the Frankencert

fuzzing tool [14] to auto-generate another 2 millions synthetic certificate chains. These chains are specially crafted certificate chains for testing certificate validation codes. First, we used OpenSSL to generate a custom root CA certificate. We then sent this custom root CA certificate, 1K real seed certificates, and a configuration file as inputs to the Frankencert tool. The configuration file is required to guide certificate generation process of Frankencert. For instance, we used the configuration file to bound the maximum number of certificate extensions to 15 and chain length to 5.

7.2 Experimental setup

We hosted our experimental setup in a Ubuntu 20.04 machine (Linux kernel 5.8.0-48) built with Intel Core i7 3.6 GHz CPU, and 32 GB of RAM. We compiled the latest versions (till January 2021) of mbedTLS, GnuTLS, and OpenSSL in this machine. Then, we ran an experimental script which validated each test certificate chain against CERES, mbedTLS, GnuTLS, and OpenSSL. During this run, we kept track of the time and chain validation outcome (with explanation) of each library for further analysis. For CERES, we validated each certificate chain twice to incorporate both PG-based parser and PC-based parser in our analysis. For mbedTLS, GnuTLS, and OpenSSL, we utilized their terminal-based certificate chain validation tools (*e.g.*, *cert_app*, *certtool*, and *openssl* respectively). For all 4 of these implementations, we relied on Linux's default trusted CA store located at "/etc/ssl/certs/ca-certificates.crt". Before testing the Frankencert generated chains, we installed the custom root CA certificate that we previously used for generating chains inside this trusted CA store. Otherwise, all the Frankencert generated chains would be *rejected* due to having untrusted root CA issuers, and we may fail to capture any other types of semantic errors.

7.3 Findings

We noticed that the chain validation outcomes of CERES were exactly same for both PG-based and PC-based parsers, giving strong empirical evidence of their *functional equivalence*. We now present the rest of our findings comparing CERES with the other 3 libraries.

7.3.1 CERES *is more restrictive.* We found CERES is more restrictive than mbedTLS, GnuTLS, and OpenSSL in terms of certificate chain validation. As shown in Table 4, CERES rejected 46.69% of Censys dataset chains and 100% of Frankencert generated chains. Overall, we observed GnuTLS and OpenSSL are more lenient than CERES and mbedTLS. The cases where CERES is more restrictive than others are thoroughly discussed in Sections 7.3.5 and 7.3.6.

Table 4: Comparison on certificate chain validation

Library	Censys (2 millions)			Frankencert (2 millions)		
	Accept	Reject	Reject %	Accept	Reject	Reject %
CERES	1,066,211	933,789	46.69%	0	2,000,000	100%
mbedTLS *v2.25.0*	1,079,342	920,658	46.03%	0	2,000,000	100%
GnuTLS *v3.6.15*	1,078,827	921,173	46.06%	513	1,999,487	99.97%
OpenSSL *v1.1.1i*	1,084,401	915,599	45.78%	648	1,999,352	99.97%

7.3.2 CERES *uniquely rejects some certificates.* We breakdown our comparison shown in Table 5 in 6 cases to obtain more insights on CERES's behavior. In the first four cases, we compared CERES with mbedTLS, GnuTLS, and OpenSSL based on the similarity of the chain validation outcomes. As an example, in case 1, we wanted to determine the number of instances when both CERES and mbedTLS rejected or accepted a particular certificate chain. In case 4, we checked whether all of these 4 implementations output the same decision for a particular chain. We learned CERES outputs the same final decision for more than 99% test certificate chains. Moreover, in the last two cases, we checked whether CERES uniquely validates any certificate chain. We found 12613 Censys dataset chains which were rejected by only CERES but were accepted by all other libraries. In contrast, there was no such instance where only CERES accepted a particular certificate chain but all others rejected that. This statistics again backs up our Finding 1.

Table 5: Comparison on similarity of outcome (accept/reject)

Comparison Case	Censys (2 millions)		Frankencert (2 millions)	
	Count	Similarity %	Count	Similarity %
CERES ∧ mbedTLS	1,986,869	99.34%	2,000,000	100%
CERES ∧ GnuTLS	1,987,384	99.37%	1,999,487	99.97%
CERES ∧ OpenSSL	1,981,792	99.09%	1,999,352	99.97%
CERES ∧ mbedTLS ∧ GnuTLS ∧ OpenSSL	1,981,789	99.09%	1,999,221	99.96%
CERES = *Reject* ∧ Others = *Accept*	12,613	0.63%	0	0%
CERES = *Accept* ∧ Others = *Reject*	0	0%	0	0%

7.3.3 *Real certificates have more semantic errors than parsing errors.* Based on CERES's chain validation outcomes and explanations, we analyzed why CERES rejected 46.69% of Censys dataset chains and 100% of Frankencert generated chains. We then found Censys dataset chains mostly had semantic errors whereas Frankencert generated chains mostly had parsing errors. As shown in Table 6, in 96.88% cases, CERES rejected Censys dataset chains due to different semantic constraint failures. This is unsurprising since these chains are real certificate chains which are expected to follow the correct X.509 certificate syntax in most cases, thus, leading to very low parsing errors. On the other hand, Frankencert certificates are generated by random mutations of 1K real certificates. Therefore, they are highly likely to contain malformed fields, and it justifies the 95.25% parsing errors for these chains.

Table 6: Categorization of errors for CERES rejected chains

Error	Censys (2 millions)			Frankencert (2 millions)		
	Count	Reject %	Details	Count	Reject %	Details
Parsing error	29,108	3.12%	Table 7	1,904,992	95.25%	Table 8
Semantic error	904,681	96.88%	Table 9	95,008	4.75%	Table 10
Total	**933,789**	**100%**	**Total**	**2,000,000**	**100%**	

∘ We now discuss some parsing errors listed in Table 7 and 8.

7.3.4 *Missing length restriction checks.* Whenever the minimum length of a particular field is explicitly mentioned in the specification (RFC 5280), CERES's Parser module strictly enforces that. For instance, the following specification quote presents the length restriction on Extensions: *"If present, Extensions is a SEQUENCE of one or more certificate extensions"*. Therefore, CERES's parser rejected 30 Censys dataset chains which had certificates with no Extension in their Extensions sequence. Similarly, length restrictions are also applicable to the *DirectoryString* choice used in *Relative Distinguished Name*; see Listing 1. Based of line 3, we rejected few certificates

which had empty (*e.g.*, "") *PrintableString* in its Issuer or Subject field. We observed mbedTLS, GnuTLS, *and* OpenSSL *do not enforce such explicitly mentioned length restrictions.*

```
DirectoryString ::= CHOICE {
    teletexString    TeletexString (SIZE (1..MAX)),
    printableString  PrintableString (SIZE (1..MAX)),
    universalString  UniversalString (SIZE (1..MAX)),
    utf8String       UTF8String (SIZE (1..MAX)),
    bmpString        BMPString (SIZE (1..MAX)) }
```

Listing 1: Structure of DirectoryString

7.3.5 Usage of deprecated IA5String in DirectoryString. Listing 1 defines the allowed string types for a *DirectoryString* structure. However, we found 28378 Censys dataset certificates still support the deprecated *IA5String* type to include "email address" in Issuer or Subject field. However, RFC 5280 restricts such identity to be included only inside Issuer Alternative Name or Subject Alternative Name extensions. We found mbedTLS, GnuTLS, *and* OpenSSL *do not prohibit usage of IA5String in DirectoryString.*

7.3.6 Incorrect encoding of UTCTime. We found 65 instances in Censys dataset chains where *UTCTime* in Validity field was encoded incorrectly. According to RFC 5280 specification, *UTCTime* must include year, month, day, hour, minute and second. However, we found those 65 certificates did not include **second** in the *UTCTime*.

7.3.7 Extra bytes in KeyUsage extension. Our analysis showed usage of extra bytes in Key Usage extension. Key Usage extension contains a bit-string to represent certificate purposes. According to ASN.1 encoding rules from section 8.6.2 of ITU-X690 [28], the encoding of a bit-string shall contain an initial octet (8 bits) followed by zero or more subsequent octets. In addition, the last of the subsequent octets must be padded with zero to seven trailing 0s, and the initial octet must encode the number of required padding bits. Now, RFC 5280 defines 9 *KeyUsage* purposes (bits). Hence, we require 3 octets **at most** to encode all 9 *KeyUsage* bits; that means, 1 octet for initial padding, and 2 octets to hold 9 bits where the last octet should contain 7 bits padding. However, we found 343 Censys dataset certificates which encoded *KeyUsage* bit-string with more than 3 octets, where the initial octet defines 0 padding, and one or more last few octets contain only 0s. CERES's parser does not allow such loose bit-string encoding. Interestingly, we found that mbedTLS, GnuTLS, *and* OpenSSL *allow these unnecessary bytes.*

7.3.8 Extensions with random octet strings. Listing 2 shows the structure of an Extension sequence. Typically, the extnValue field is an octet string which contains some nested fields in ASN.1 encoding. Our analysis revealed more than 90% of the Frankencert certificates were generated with one or more Extensions which were holding just some random bytes (*i.e.*, not further parseable) in extnValue field. These test certificates allowed us to figure out a leniency in GnuTLS. According to RFC 5280, a certificate must be rejected if it contains a critical Extension that the implementation cannot recognize. Now, CERES recognizes the top 10 most frequent extensions in Table 2, and if any of those extensions do not hold correct extnValue octet string structure, we report parsing errors. Our analysis on discrepancies showed OpenSSL accepts random octet strings for at least Certificate Policies extension and Authority Information Access extension. Moreover, we found GnuTLS *accepts random octet string for any Extension, even for the critical ones.*

```
Extension ::=  SEQUENCE {
    extnID      OBJECT IDENTIFIER,
    critical    BOOLEAN DEFAULT FALSE,
    extnValue   OCTET STRING
        --  contains the DER encoding of
       an ASN.1 value corresponding to the
       extension type identified by extnID }
```

Listing 2: Structure of an Extension

Table 7: Parsing errors detected in Censys chains

Field	Reason	Count	Perc.
Subject	$Length(string) = 0$	6	0.02%
	Deprecated *IA5String* type	2,296	7.89%
Issuer	$Length(string) = 0$	22	0.08%
	Deprecated *IA5String* type	26,082	89.60%
Validity	Incorrect *UTCTime* encoding	65	0.22%
Extensions	$Length(sequence) = 0$	30	0.10%
Subject Alternative Name	$Length(sequence) = 0$	146	0.50%
Issuer Alternative Name	$Length(sequence) = 0$	14	0.05%
Key Usage Extension	Loose bit-string encoding	343	1.18%
Authority Key Identifier Extension	$Length(Identifier) = 0$	103	0.36%
RSA Signature	Missing block type	1	$\approx 0\%$
	Total	**29,108**	**100%**

Table 8: Parsing errors detected in Frankencert chains

Field	Reason	Count	Perc.
Subject	$Length(string) = 0$	11,222	0.59%
	Deprecated *IA5String* type	89,666	4.71%
Issuer	$Length(string) = 0$	12,180	0.64%
	Deprecated *IA5String* type	89,323	4.69%
Subject Alternative Name	Random Octet String	81,548	4.28%
Issuer Alternative Name	Random Octet String	301,929	15.85%
Basic Constraints	Random Octet String	255,579	13.42%
Extended Key Usage	Random Octet String	254,940	13.38%
Key Usage	Random Octet String	253,186	13.29%
Authority Key Identifier	Random Octet String	109,686	5.76%
Subject Key Identifier	Random Octet String	81,761	4.29%
Certificate Policies	Random Octet String	128,778	6.75%
Authority Info. Access	Random Octet String	132,311	6.95%
CRL Distribution Points	Random Octet String	101,575	5.33%
RSA Signature	Missing block type	1,299	0.07%
	Incorrect padding bytes	9	$\approx 0\%$
	Total	**1,904,992**	**100%**

∘ We now discuss few violations of semantic constraint based on Tables 9 and 10.

7.3.9 Supporting unknown certificate version (SCP2 + SCP3). RFC 5280 specification allows only Version 1, 2, and 3 certificates (SCP3). Additionally, it restricts usage of Extensions only to Version 3 certificates (SCP2). Although CERES implements these checks, our analysis on validation outcomes of Frankencert generated chains revealed that GnuTLS *and* OpenSSL *do not enforce these checks properly.* For example, we surprisingly found a few instances where both these libraries allowed unsupported Version 4 certificates which also had Extensions. These certificates simultaneously violate SCP2 and SCP3 constraints. In addition, OpenSSL allowed Version 1 certificates with Extensions, which GnuTLS, however, correctly rejected.

7.3.10 Allowing zero (0) as serial number (SCP4). As discussed in Appendix B, there is an inconsistency in RFC 5280 regarding the values allowed for Serial number. Our interpretation is that "Conforming CAs must issue certificate with Serial > 0". We found 11162 Censys dataset certificates which had 0 as Serial number; hence, we rejected those. However, we noticed mbedTLS, GnuTLS, *and* OpenSSL *did not reject 0 as certificate Serial number.*

7.3.11 Rejecting unknown critical extension (SCP15). We noticed a significant number of Censys dataset certificates had Certificate Transparency Poison extension as a critical extension. Since this extension is not a standard extension, CERES processed this as an unknown extension and eventually rejected the certificates.

7.3.12 Incorrect usage of certificate purposes (SCP13 + CCP1). We found few non-CA certificates which violated the constraint SCP13 due to having KeyCertSign key usage bit on. In addition, we checked whether combinations of Extended Key Usage and Key Usage purposes are RFC 5280 compliant. For instance, ServerAuth Extended Key Usage purpose is consistent only with DigitalSignature, or KeyAgreement Key Usage purposes.

7.3.13 Incorrect name chaining (CCP6). For certificate chain path validation, name chaining is performed by matching the Issuer name in one certificate with the Subject name in its issuer CA certificate. Rules for this name matching is described in Section 7.1 of RFC 5280. We found 11 Censys dataset chains which did not follow the name matching rules.

7.3.14 Allowing insecure RSA signature (CCP8). CERES supports only RSA signature verification and prohibits any weak or deprecated signature hash algorithms (*e.g.*, MD5, SHA1), or any RSA public key shorter than 2048 bits. We observed the presence of 1024 bits RSA public keys in 77094 Frankencert generated chains; hence, those chains were rejected by CERES. In contrast, we noticed GnuTLS *and* OpenSSL *allowed 1024 bits RSA public keys.*

Table 9: Semantic errors detected in Censys chains

SCP = Single Certificate Property CCP = Chain Certificate Property

Constraint	Reason	Count	Perc.
SCP1	Signature algorithm mismatch	1	≈ 0%
SCP2	Extensions present, but Version ≠ 3	41	≈ 0%
SCP3	Unsupported certificate Version 4 (value 3)	23	≈ 0%
SCP4	Serial = 0	11,162	1.23%
SCP13	KeyCertSign = True in non-CA certificate	12	≈ 0%
SCP14	Repeated Extension	1	≈ 0%
SCP15	Unknown critical Extension	737,984	81.57%
SCP18	Expired certificate	631	0.07%
CCP1	Inconsistent certificate purpose	3	≈ 0%
CCP4	DistributionPoint has CRLIssuer though issuer is CRL issuer	6	≈ 0%
CCP6	Incorrect name chaining	11	≈ 0%
CCP7	Untrusted root CA issuer	149,418	16.52%
CCP8	RSA signature verification failure	1	≈ 0%
	Insecure RSA signature hash algorithm (SHA1)	5,385	0.61%
	Short RSA public key (1024 bits)	2	≈ 0%
	Total	**904,681**	**100%**

7.4 Runtime analysis

For Censys dataset chains, we tracked runtime for all four libraries. We also measured runtime for the Parser, and Semantic-checker modules of CERES to breakdown its overall chain validation times. The upper half of Table 11 shows comparison of CERES with other

Table 10: Semantic errors detected in Frankencert chains

SCP = Single Certificate Property CCP = Chain Certificate Property

Constraint	Reason	Count	Perc.
SCP2	Extensions present, but Version ≠ 3	3,260	3.43%
SCP13	KeyCertSign = True in non-CA certificate	82	0.09%
SCP15	Unknown critical Extension	14,562	15.33%
CCP1	Inconsistent certificate purpose	10	0.01%
CCP8	Short RSA public key (1024 bits)	77,094	81.14%
	Total	**95,008**	**100%**

Table 11: Chain verification runtime (second) analysis

Library	Min	Max	Mean	Median	Standard Deviation
mbedTLS	0.036	0.084	0.042	0.041	0.003
GnuTLS	0.057	0.084	0.064	0.064	0.002
OpenSSL	0.042	0.064	0.047	0.046	0.002
CERES (PC-based)	0.105	0.696	0.462	0.467	0.074
CERES (PG-based)	0.105	0.890	0.583	0.582	0.106
Detailed Time Complexity of CERES					
Module	**Min**	**Max**	**Mean**	**Median**	**Standard Deviation**
PC-based Parser	0.001	0.099	0.028	0.026	0.009
PG-based Parser	0.001	0.363	0.163	0.156	0.051
Semantic-checker	0.000	0.672	0.389	0.395	0.080

libraries. We found CERES requires 0.462s and 0.583s on average when PC-based and PG-based parsers are used, respectively. The lower half of Table 11 shows breakdown of CERES's runtime. We noticed that the Semantic-checker module takes 0.389s on average due to the use of expensive SMT solver. Though CERES requires comparatively longer times than other libraries, it is still reasonably fast in real time. *Note that*, the runtime of Semantic-checker mostly depends on maximum chain length and maximum number of Extensions that we considered to encode the semantic constraints. In our evaluation, we set maximum chain length to 5, and number of Extensions to 15, which are reasonable parameters in reality.

8 DISCUSSION

8.1 Takeaway for application developers

Use case. CERES is envisioned to be used as an oracle for checking noncompliance of a given library implementation. One can also use CERES to validate a chain to be used for configuring a TLS-enabled webserver. Although not one of its envisioned use case, application developers can directly use CERES in their applications to delegate the task of X.509 certificate chain validation.

Modular decomposition. Our decomposition of CERES in different modules has several advantages. (1) the overall source code is relatively cleaner and easy to follow. (2) It also allows us to independently debug and extend any module.

Parser selection. PG-based parser is comparatively slower due to more complex source code. According to Table 11, the PC-based parser spends 0.028 sec on average for parsing, whereas the PG-based parser takes 0.163 sec on average. Therefore, CERES by default selects the PC-based parser.

8.2 Threat to validity

We follow a best-effort approach to manually interpret the standard and translate it into a QFFOL formula. Although our empirical evaluation gives confidence about our interpretation's correctness, we do not claim our interpretation to be faithful to designers' intent.

Hence, our interpretation should not be considered as the official interpretation intended by the RFC authors.

Our measurement of extensions are performed on a Censys dataset snapshot from 2019. Theoretically, the frequencies of the extensions from a current snapshot may vary but we do not expect the difference to be significant.

8.3 Limitations

We want to emphasize that CERES has not been formally proven to be functionally correct with respect to the standard. This is why we refrain from referring to CERES as the reference implementation and instead refer to it as a high-assurance implementation. The main contribution of this paper is the discipline of separating the requirements into syntactic and semantic ones, and developing an executable specification for them. As such, we do not claim our formalization to be complete (*i.e.*, containing all the applicable syntactic and semantic rules). Our specification is, however, modular enough so that newer constraints can be easily added.

As discussed in Section 6.4.2, we currently only support RSA signature verification. For RSA signature verification, we rely on Python to compute *hash digests* (via its hashlib library) and *modular exponentiation*. Currently, we also do not check certificate revocation status and do not match hostnames. There is no clear direction whether to include these checks in X.509 implementations, or to leave these tasks for application developers [14]. The task of validating certificate chain can be separated from hostname checking, as assumed by previous work [15, 42]. Leaving these out simplifies the theories used in SMT and enables proof-generation. Note that, the proofs generated by the SMT solver [12] to demonstrate the noncompliance can have holes in them, especially, lacking proofs for arithmetic lemmas and rewrite steps. An upcoming version of the CVC4 SMT solver [12], however, addresses these holes.

8.4 From Specification to Implementation

CERES on average incurs an $11x$ runtime overhead compared to tested implementations, making it difficult to be used as a drop-in replacement. This slowdown can be mainly attributed to the call to the SMT solver. One may naturally ask whether it is possible to avoid this SMT-related runtime overhead by developing an implementation completely in a language like C or Rust. The answer to this question is yes. However, one should ask what kind of guarantees (*i.e.*, formally proven correct or not) would one expect from such an implementation. Based on this answer, one can envision moving from the formal specification developed in this paper to an implementation in C/Rust in the following two ways.

Unverified C/Rust implementation. Recall that, an X.509 implementation has the following four components: (1) a parser; (2) an input transformer (*e.g.*, chain building, string canonicalization, cryptographic operations for verifying digital signatures); (3) a semantic checker; (4) a driver that connects the three aforementioned components. For generating a not-formally-verified implementation in C/Rust from CERES' specification, one has to do the following: (1) modify CERES' DSL parser generator to spit a parser in C/Rust; (2) rewrite the input transformer in C/Rust; (2) convert each QFFOL assertion in the semantic restrictions into one or more if-then-else statements in C/Rust; (4) rewrite the driver in C/Rust.

Verified C/Rust implementation. Achieving a formally verified X.509 implementation from CERES' specification, however, would require formally proving the correctness (*i.e.*, soundness, completeness, and termination) and possibly, other properties (*e.g.*, memory safety) of all the four CERES components. One can use deductive verifiers such as FramaC [20] (for C) or Electrolysis [45] (for Rust) for formally proving the desired correctness guarantees. Such verification, however, requires nontrivial human efforts of (1) decomposing the specification into individual function contracts, and (2) coming up with auxiliary lemmas for the proofs to go through.

Our grand vision is having a fully verified C/Rust implementation eventually. This paper's formalization and re-engineering efforts are foundational steps towards that goal. The main point we want to convey with CERES is that it is indeed possible to enforce a useful portion of the X.509 specification with an SMT solver. Due to the huge manual effort that is required to develop a formally verified X.509 implementation and also to limit this paper's scope, we leave the grand vision as future work.

8.5 Responsible disclosure

We initially disclosed our findings to the corresponding library developers in *private*. GnuTLS developers positively acknowledged our findings, and promised to fix the reported noncompliance in their later releases. In contrast, OpenSSL and mbedTLS developers did not acknowledge our findings claiming the reported noncompliance instances *are not security sensitive*. This reflects the reluctance of the library developers on developing RFC compliant X.509 implementations. However, we still submitted *public* bug reports to OpenSSL and mbedTLS developers with some expectations on removing those noncompliance instances in their future releases.

9 CONCLUSION

In this paper, we re-engineer and formalize the X.509 standard specification alleviating its design complexity, ambiguities, or underspecifications, and then use it to develop a high-assurance implementation CERES. After separating the syntactic and semantic requirements from the X.509 specification, we formalize the syntactic requirements based on a restricted-fragment of attribute grammar. In contrast, we encode the semantic requirements using QFFOL formula, which results in an executable specification that can be efficiently enforced by an SMT solver. We compared CERES with 3 mainstream libraries based on 4 million real and synthetic certificate chains, and observed that CERES rightfully rejects malformed and invalid certificates. We also observed some cases where other libraries do not reject non-compliant certificates. We have responsibly disclosed our findings to the corresponding library developers and received positive acknowledgment from the maintainer of GnuTLS.

Acknowledgment

We thank the anonymous reviewers for their detailed and insightful comments. We are grateful to Alan Mislove and Cesare Tinelli for their helpful discussions. The work reported here was supported in part by the NSF Grant CNS-2006556, and serves as the foundation of a grant from the Research Grants Council (RGC) of Hong Kong (Project No.: CUHK 24205021).

REFERENCES

[1] 2014. stringprep: Implements the "StringPrep" algorithm. https://hackage.haskell.org/package/stringprep.

[2] 2020. SSL certificate chain resolver. https://github.com/zakjan/cert-chain-resolver.

[3] 2021. GnuTLS. https://www.gnutls.org/.

[4] 2021. mbedTLS. https://tls.mbed.org/.

[5] 2021. OpenSSL. https://www.openssl.org/.

[6] 2021. parsec 3.8. https://pypi.org/project/parsec/.

[7] Carlisle Adams and Steve Lloyd. 2003. *Understanding PKI: concepts, standards, and deployment considerations.* Addison-Wesley Professional.

[8] Henk Alblas and Borivoj Melichar. 1991. *Attribute Grammars, Applications and Systems: International Summer School SAGA, Prague, Czechoslovakia, June 4-13, 1991. Proceedings.* Vol. 545. Springer Science & Business Media.

[9] Julian Bangert and Nickolai Zeldovich. 2014. Nail: A Practical Tool for Parsing and Generating Data Formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. 615–628.

[10] Alessandro Barenghi, Nicholas Mainardi, and Gerardo Pelosi. 2018. Systematic parsing of X. 509: eradicating security issues with a parse tree. *Journal of Computer Security* (2018), 817–849.

[11] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *SMT Workshop*.

[12] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification (Lecture Notes in Computer Science)*. 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

[13] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th USENIX Security Symposium (SEC'17)*. 917–934.

[14] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security and Privacy (SP)*. 114–129.

[15] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*. 503–520. https://doi.org/10.1109/SP.2017.40

[16] Sze Yiu Chau, Moosa Yahyazadeh, Omar Chowdhury, Aniket Kate, and Ninghui Li. 2019. Analyzing semantic correctness with symbolic execution: A case study on pkcs# 1 v1. 5 signature verification. In *Network and Distributed Systems Security Symposium 2019 (NDSS '19)*.

[17] Yuting Chen and Zhendong Su. 2015. Guided Differential Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. 793–804. https://doi.org/10.1145/2786805.2786835

[18] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. 2011. Computing Small Unsatisfiable Cores in Satisfiability modulo Theories. *J. Artif. Int. Res.* (2011), 701–728.

[19] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. 2008. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* Technical Report. http://www.rfc-editor.org/rfc/rfc5280.txt

[20] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods (SEFM'12)*. 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

[21] Joyanta Debnath, Sze Yiu Chau, and Omar Chowdhury. 2021. Source Code for CERES. https://github.com/joyantaDebnath/CERES.

[22] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats. *Proc. ACM Program. Lang.* 3, ICFP, Article 82 (July 2019), 29 pages. https://doi.org/10.1145/3341686

[23] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. 2015. A Search Engine Backed by Internet-Wide Scanning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. 542–553. https://doi.org/10.1145/2810103.2813703

[24] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. 2013. Analysis of the HTTPS Certificate Ecosystem. In *Proceedings of the 2013 ACM SIGCOMM Internet Measurement Conference (IMC '13)*. 291–304. https://doi.org/10.1145/2504730.2504755

[25] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. 2010. The next 700 Data Description Languages. *J. ACM* 57, 2, Article 10 (Feb. 2010), 51 pages. https://doi.org/10.1145/1667053.1667059

[26] Marco Gario and Andrea Micheli. 2015. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop*.

[27] P. Hoffman and M. Blanchet. 2002. *Preparation of Internationalized Strings ("stringprep").* Technical Report. https://tools.ietf.org/rfc/rfc3454.txt

[28] ITU-T. 2021. X.690 : Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). (2021).

[29] Trevor Jim, Yitzhak Mandelbaum, and David Walker. 2010. Semantics and Algorithms for Data-Dependent Grammars. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. 417–430. https://doi.org/10.1145/1706299.1706347

[30] Burt Kaliski. 1998. *PKCS# 1: RSA encryption version 1.5.* Technical Report. https://tools.ietf.org/rfc/rfc2313.txt

[31] David Kaloper-Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. 2015. Not-Quite-So-Broken TLS: Lessons in Re-Engineering a Security Protocol Specification and Implementation. In *24th USENIX Security Symposium (SEC'15)*. 223–238.

[32] Dan Kaminsky, Meredith L. Patterson, and Len Sassaman. 2010. PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure. In *Financial Cryptography and Data Security*. 289–303.

[33] Donald E. Knuth. 1968. Semantics of Context-Free Languages. In *In Mathematical Systems Theory*. 127–145.

[34] Ulrich Kühn, A. Pyshkin, E. Tews, and R. Weinmann. 2008. Variants of Bleichenbacher's Low-Exponent Attack on PKCS#1 RSA Signatures. (2008).

[35] Deepak Kumar, Zhengping Wang, Matthew Hyder, Joseph Dickinson, Gabrielle Beck, David Adrian, Joshua Mason, Zakir Durumeric, J. Alex Halderman, and Michael Bailey. 2018. Tracking Certificate Misissuance in the Wild. In *2018 IEEE Symposium on Security and Privacy (SP)*. 785–798. https://doi.org/10.1109/SP.2018.00015

[36] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. 2007. PADS/ML: A Functional Data Description Language. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 77–83. https://doi.org/10.1145/1190216.1190231

[37] Duckki Oe, Andrew Reynolds, and Aaron Stump. 2009. Fast and Flexible Proof Checking for SMT. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories* (Montreal, Canada) *(SMT '09)*. Association for Computing Machinery, New York, NY, USA, 6–13. https://doi.org/10.1145/1670412.1670414

[38] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. 2006. Binpac: A yacc for writing application protocol parsers. In *Proceedings of the 2006 ACM SIGCOMM Internet Measurement Conference (IMC '06)*. 289–300. https://doi.org/10.1145/1177080.1177119

[39] Terence Parr. 2013. *The definitive ANTLR 4 reference.* Pragmatic Bookshelf.

[40] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Beguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *2020 IEEE Symposium on Security and Privacy (SP)*. 983–1002. https://doi.org/10.1109/SP40000.2020.00114

[41] Tahina Ramananandro, Antoine Delignat-Lavaud, Cedric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *28th USENIX Security Symposium (SEC'19)*. 1465–1482.

[42] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. 2017. HVLearn: Automated Black-Box Analysis of Hostname Verification in SSL/TLS Implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*. 521–538. https://doi.org/10.1109/SP.2017.46

[43] Robin Sommer, Johanna Amann, and Seth Hall. 2016. Spicy: A Unified Deep Packet Inspection Framework for Safely Dissecting All Your Data. In *Annual Conference on Computer Security Applications (ACSAC '16)*. 558–569. https://doi.org/10.1145/2991079.2991100

[44] Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V. Thakur. 2021. DICE*: A Formally Verified Implementation of DICE Measured Boot. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1091–1107. https://www.usenix.org/conference/usenixsecurity21/presentation/tao

[45] Sebastian Andreas Ullrich. 2016. Simple Verification of Rust Programs via Functional Purification.

[46] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 193 (Nov. 2020), 25 pages. https://doi.org/10.1145/3428261

[47] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. 718–730. https://doi.org/10.1145/3385412.3385985

[48] K Zeilenga. 2006. *Lightweight Directory Access Protocol (LDAP): Internationalized String Preparation.* Technical Report. https://www.rfc-editor.org/rfc/rfc4518.txt

[49] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In

Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17). 1789–1806. https://doi.org/10.1145/3133956.3134043

A SYNTACTIC REQUIREMENTS IN OUR DSL

```
{ # exchange context from children to parent
  attribute-of-parent <- # Arithmetic expression of
       attributes-of-Children
  .
  .
  .

  # pre-condition to handle choices
  PRECONDENT : # boolean expression

  # pre-condition to handle the base case of a
  # recursive grammar rule
  PRECONDRET : # boolean expression

  # post-condition to check after parsing all child
       fields
  POSTCOND : # boolean expression
}
```

Listing 3: Representation of computation-block

```
<Parent>(context) ::= <Child-1>(context) <Child-2>
{ ... } # computation-block
```

Listing 4: Context passing from parent to child

```
<Parent> ::= <Child-1> <Child-2>(context-from-Child-1)
{ ... }
```

Listing 5: Context passing among children (left to right)

```
<Parent> ::= <Child-1> <Child-2>
{ Attrbt1.Parent <- Attrbt1.Child-1 + Attrbt2.Child-2
  .
  .
  .
  Attrbtn.Parent <- Attrbtn.Child-1 - Attrbtn.Child-2
}
```

Listing 6: Context passing from children to parent

Listings 3, 4, 5, 6, 7, and 8 show how to use our DSL for capturing the syntactic requirements. We then show some examples of how to capture the requirements of an X.509 certificate. Each grammar rule starts with the pattern $A ::= B_1\ B_2\ ...\ B_n$ followed by a computation-block {...}, where A is the head of the current production rule (referred to as the parent field), and $B_1\ B_2\ ...\ B_n$ are non-terminals used to define A (referred to as the children fields). The computation-block (see Listing 3) is intended to provide ways to exchange contextual information (semantic information) from the children to their parent (*i.e.*, computation of synthesized attribute) as well as to check certain pre-condition and/or post-condition over some context. The pre-condition is useful to decide whether the child fields should be parsed based on the given context, and the post-condition can be used to check certain semantic requirements of the already parsed child fields. Listings 3, 4, 5, and 6 show how to exchange contextual information among parent and/or child fields.

```
<Parent> ::= <Child-1>_? <Child-2>
{ ... }
```

Listing 7: Representation of Optional child node

```
<Parent> ::= <Child-1> <Child-2> # choice1
{ ... } # computation-block for choice1
 | <Child-A> <Child-B>
{ ... }
 .
 .
 .
 | <Child-X> <Child-Y>
{ ... }
```

Listing 8: Representation of Choice

Listings 7 and 8 show representation of *optional* field (using _?) and *choices* (using |), respectively.

```
1  Certificate ::= Type Length FCert
2   { SIZE_Cert <- SIZE_Type + SIZE_Length + SIZE_FCert
3     POSTCOND : SIZE_FCert = VAL_Length && VAL_Type = 48 }
4
5  Serial ::= Type Length Value(VAL_Length)
6   { SIZE_Serial <- SIZE_Type + SIZE_Length + SIZE_Value
7     POSTCOND : SIZE_Value = VAL_Length }
8
9  Value(VAL_Length) ::= Byte Value(VAL_Length - 1)
10     { SIZE_Value <- SIZE_Byte + SIZE_Value
11       PRECONDRET : VAL_Length > 0 }
12
13 Extnsopts(VAL_Extnid) ::= Fieldsaki(VAL_Extnid)
14  { .
15    .
16    PRECONDENT : VAL_Extnid = 5578019 }
17  | Fieldsski(VAL_Extnid)
18  { .
19    .
20    PRECONDENT : VAL_Extnid = 5577998 }
```

Listing 9: Example on X.509 grammar

In listing 9, we present a few examples of the X.509 grammar expressed in our DSL, where *SIZE* is a synthesized attribute which calculates length of the parent field based on the length of its children; see lines 2, 6, and 10. Lines 3 and 7 show how to use *POSTCOND* (*i.e.*, post-condition) to check length constraints of ASN.1 $\langle t, \ell, v \rangle$ form. Line 11 shows the usage of *PRECONDRET* to express the base condition of the recursively defined *Value* field. The pre-condition defined in *PRECONDRET* is checked before parsing the first child field *Byte*. This construct enables us to specify when to stop parsing a recursive field. That means, we ensure that the number of bytes parsed under the *Value* field is bounded by the value of *Length* field (line 5).

The pre-conditions defined using *PRECONDENT* are useful for selecting the appropriate expansion rule in case of *choices* in the grammar rule (see lines 13–20). Similar to *PRECONDRET*, it also checks the pre-condition before parsing the first child (*e.g.*, *Fieldsaki*, *Fieldsski*). This is analogous to having access to the lookahead to decide which choice to pursue and thus avoiding the need for backtracking.

B EXAMPLES OF INCONSISTENCY, AMBIGUITY, AND UNDER-SPECIFICATION

Perhaps unsurprisingly, the specification documents considered are all written in English, which is a natural language that is prone to inconsistency, ambiguity and misinterpretation, and we have indeed observed several instances of problematic clauses in the RFC 5280. Here we give a few illustrative examples.

Regarding the requirements on serial number, in Section 4.1.2.2, RFC 5280 says:

> *"The serial number MUST be a positive integer assigned by the CA to each certificate...CAs MUST force the serial Number to be a non-negative integer...Non-conforming CAs may issue certificates with serial numbers that are negative or zero. Certificate users SHOULD be prepared to gracefully handle such certificates."*

The first sentence is inconsistent with the last sentence: one excludes zero, while the other allows it. An errata on this has been filed [1] back in 2012 but at the time of writing it this does not seem to be included in any RFC updates or clarifications.

We now give an example of requirements that we consider to be ambiguous. Regarding the contents of the CRL distribution points extension, in Section 4.2.1.13, RFC 5280 says:

> *" A DistributionPoint consists of three fields, each of which is optional: distributionPoint, reasons, and cRLIssuer. While each of these fields is optional, a DistributionPoint MUST NOT consist of only the reasons field; either distributionPoint or cRLIssuer MUST be present. If the certificate issuer is not the CRL issuer, then the cRLIssuer field MUST be present and contain the Name of the CRL issuer. If the certificate issuer is also the CRL issuer, then conforming CAs MUST omit the cRLIssuer field and MUST include the distributionPoint field. "*

However, it is not immediately clear whether the *either ... or ...* clause should be interpreted as an *exclusive or* ($\oplus$), or should it be represented with a *logical or* ($\vee$). If one assumes the *exclusive or* interpretation, then the *MUST omit* clause in the last sentence of the quote seems to be somewhat redundant, as it is already implied by the later *MUST include* clause of the same sentence. However, if *logical or* is indeed the correct interpretation, that is, it is acceptable for both `distributionPoint` and `cRLIssuer` to be present, then the *either ... or ...* could have been written as *at least one of ... and ...* to make it less confusing. Such interpretation matters because it outlines the correct combinations of the fields that constitute the CRL distribution points extension, and affects what should be deemed as syntactically correct by the parser.

Additional, we argue that RFC 5280 also suffers from the problem of under-specification. Many clauses concerning the choice of values and options are stipulated as *producer rules* (*e.g.*, *conforming CAs must ...*), but it is not immediately apparent whether some or all of these should also be enforced by the consumer. In some cases, RFC 5280 mentioned that certificate user should *gracefully handle* certain non-conforming inputs, without really defining what needs to happen. Does that mean the validation should not crash? Should the non-conforming inputs be rejected or processed as normal? RFC 5280 is not explicit on those questions. Similarly, it also did not make clear on what should the certificate user do in cases where the certificate itself violates the DER.

C LIST OF SEMANTIC REQUIREMENTS

The list of semantic rules considered by CERES are presented in Tables 12 and 13.

[1] https://www.rfc-editor.org/errata/eid3200

Table 12: Semantic properties for a single X.509 certificate.

SCP = Single Certificate Property

Constraint	Description
SCP1	SignatureAlgorithm field MUST contain the same algorithm identifier as the Signature field in the sequence TbsCertificate.
SCP2	Extension field MUST only appear if the Version is 3 .
SCP3	At a minimum, conforming implementations MUST recognize Version 3 certificates. Generation of Version 2 certificates is not expected by implementations based on this profile.
SCP4	The Serial number MUST be a positive integer assigned by the CA to each certificate. Certificate users MUST be able to handle Serial number values up to 20 octets.
SCP5	The Issuer field MUST contain a non-empty distinguished name (DN).
SCP6	If the Subject is a CA (*e.g.*, the Basic Constraints extension, is present and the value of CA is TRUE), then the Subject field MUST be populated with a non-empty distinguished name.
SCP7	Unique Identifiers fields MUST only appear if the Version is 2 or 3. These fields MUST NOT appear if the Version is 1.
SCP8	Where it appears, the PathLenConstraint field MUST be greater than or equal to zero.
SCP9	If the Subject is a CRL issuer (*e.g.*, the Key Usage extension, is present and the value of CRLSign is TRUE), then the Subject field MUST be populated with a non-empty distinguished name.
SCP10	When the Key Usage extension appears in a certificate, at least one of the bits MUST be set to 1.
SCP11	If subject naming information is present only in the Subject Alternative Name extension , then the Subject name MUST be an empty sequence and the Subject Alternative Name extension MUST be critical.
SCP12	If the Subject Alternative Name extension is present, the sequence MUST contain at least one entry.
SCP13	If the KeyCertSign bit is asserted, then the CA bit in the Basic Constraints extension MUST also be asserted. If the CA boolean is not asserted, then the KeyCertSign bit in the Key Usage extension MUST NOT be asserted.
SCP14	A certificate MUST NOT include more than one instance of a particular Extension.
SCP15	A certificate-using system MUST reject the certificate if it encounters a critical Extension it does not recognize or a critical Extension that contains information that it cannot process.
SCP16	A certificate policy OID MUST NOT appear more than once in a Certificate Policies extension.
SCP17	While each of these fields is optional, a DistributionPoint MUST NOT consist of only the Reasons field; either distributionPoint or CRLIssuer MUST be present.
SCP18	The certificate Validity period includes the current time.

Table 13: Semantic properties for a chain of X.509 certificates

CCP = Certificate Chain Property

Constraint	Description
CCP1	If a certificate contains both a Key Usage extension and an Extended Key Usage extension, then both extensions MUST be processed independently and the certificate MUST only be used for a purpose consistent with both extensions. If there is no purpose consistent with both extensions, then the certificate MUST NOT be used for any purpose.
CCP2	Conforming implementations may choose to reject all Version 1 and Version 2 intermediate CA certificates .
CCP3	The PathLenConstraint field is meaningful only if the CA boolean is asserted and the Key Usage extension, if present, asserts the KeyCertSign bit. In this case, it gives the maximum number of non-self-issued intermediate certificates that may follow this certificate in a valid certification path.
CCP4	For DistributionPoint field, if the certificate issuer is not the CRL issuer, then the CRLIssuer field MUST be present and contain the Name of the CRL issuer. If the certificate issuer is also the CRL issuer, then conforming CAs MUST omit the CRLIssuer field and MUST include the distributionPoint field.
CCP5	A certificate MUST NOT appear more than once in a prospective certification path.
CCP6	Certificate users MUST be prepared to process the Issuer distinguished name and Subject distinguished name fields to perform name chaining for certification path validation.
CCP7	Validate whether root CA certificate is trusted by system.
CCP8	Validate RSA signatures.
CCP9	Validate leaf certificate purpose against user's expected certificate purpose.
CCP10	Every non-leaf certificate in a chain must be a CA certificate.